A Comparative Study on Streaming Frameworks for Big Data

Wissem Inoubli, Sabeur Aridhi, Haithem Mezni, Mondher Maddouri, Engelbert Mephu Nguifo

A Comparative Study on Streaming Frameworks for Big Data

Wissem Inoubli [1], **Sabeur Aridhi**[2], **Haithem Mezni**[3], **Mondher Maddouri**[4], **Engelbert Mephu Nguifo**[5]

[1]University of Tunis El Manar, Faculty of Sciences of Tunis, LIPAH, Tunis, Tunisia

[2]University of Lorraine, CNRS, Inria, LORIA, F-54000 Nancy, France

[3]University of Jendouba, SMART Lab, Jendouba, Tunisia

[4]College Of Buisness, University of Jeddah, P.O.Box 80327, Jeddah 21589 KSA

[5]University of Clermont Auvergne, LIMOS, Clermont-Ferrand, France

Abstract. *Recently, increasingly large amounts of data are generated from a variety of sources. Existing data processing technologies are not suitable to cope with the huge amounts of generated data. Yet, many research works focus on streaming in Big Data, a task referring to the processing of massive volumes of structured/unstructured streaming data. Recently proposed streaming frameworks for Big Data applications help to store, analyze and process the continuously captured data. In this paper, we discuss the challenges of Big Data and we survey existing streaming frameworks for Big Data. We also present an experimental evaluation and a comparative study of the most popular streaming platforms.*

1. Introduction

In recent decades, increasingly large amounts of data are generated from a variety of sources. The size of generated data per day on the Internet has already exceeded two exabytes [6]. Stream processing problems lead to several research questions such as (1) how to design scalable environments, (2) how to provide fault tolerance and (3) how to design efficient solutions. In this context, stream processing frameworks are mainly designed to process the huge amount of data streams and to make on-the-fly decisions. With the rise of big data, various organizations have started to employ stream frameworks to solve major emerging big data problems related to smart ecosystems, healthcare services, social media, etc. For example, in smart cities, various sensors such as GPS, weather conditions devices, public transportation smart cards and traffic cameras are installed on diverse regions (e.g, water lines, utility poles, buses, trains, traffic lights) [10]. From these sensors, very large quantities of data are collected. To understand such volume of data, it is important to reveal hidden and valuable information from the big stream/storage of data. Social media is another representative data source for big data that requires real-time processing and results [13]. In fact, a huge volume of data is instantly and continuously generated from a wide range of Internet applications and Web sites. Examples include online mobile photo and video sharing services (e.g, Instagram, Youtube, Flickr), social networks (e.g. Facebook, Twitter), business-oriented networks (e.g. LinkedIn), etc. The adoption of in-stream frameworks that offer iterative processing and learning capabilities allows to effectively perform specific tasks such as social network analysis, links prediction, etc. Given the importance of the above discussed real-world scenarios, finding the

relevant framework for the big/high stream-oriented applications becomes a challenging problem.

Several systems have been proposed in the literature. In this paper, we present a comparative study of popular stream processing frameworks according to their key features. The studied frameworks have been chosen based on their number of contributors. Our contributions are summarized as follows:

- We present four popular streaming frameworks for big data, their architecture and their internal behavior.
- We compare the presented frameworks according to their key features.
- We evaluate the performance of the presented frameworks in terms of resources consumption.

This paper is organized as follows. In Section 2, we present some well-known stream frameworks for big data. Section 3 is devoted to the comparison of the discussed frameworks. In Section 4, an experimental evaluation of the presented stream processing frameworks is provided.

2. Stream processing frameworks for big data

Several streaming frameworks for big data have been proposed to allow real-time large-scale stream processing. This section sheds the light on the most popular big data stream processing frameworks and provides a comparison study of them according to their main features.

2.1. Apache Spark

Apache Spark [11] is a powerful processing framework that provides an ease of use tool for efficient analytics of heterogeneous data. It was originally developed at UC Berkeley in 2009 [14] . Spark has several advantages compared to other big data frameworks like Hadoop MapReduce [4] and Storm [12]. A key concept of Spark is Resilient Distributed Datasets (RDDs). An RDD is basically an immutable collection of objects spread across a Spark cluster. In Spark, there are two types of operations on RDDs: (1) transformations and (2) actions. Transformations consist in the creation of new RDDs from existing ones using functions like map, filter, union and join. Actions consist of final result of RDD computations. Spark Streaming is a Spark library that enables scalable and high-throughput stream processing of live data streams.

2.2. Apache Storm

Storm [12] is an open source framework for processing large structured and unstructured data in real time. Storm is a fault tolerant framework that is suitable for real time data analysis, machine learning, sequential and iterative computation. A Storm program is represented by a directed acyclic graphs (DAG). The edges of the program DAG represent data transfer. The nodes of the DAG are divided into two types: spouts and bolts. The spouts (or entry points) of a Storm program represent the data sources. The bolts represent the functions to be performed on the data. Note that Storm distributes bolts across multiple nodes to process the data in parallel. Storm is based on two daemons called Nimbus (in master node) and a supervisor for each slave node. Nimbus supervises the slave nodes and assigns tasks to them. If it detects a node failure in the cluster, it reassigns

the task to another node. Each supervisor controls the execution of its tasks (affected by the nimbus). It can stop or start the spots following the instructions of Nimbus. Each topology submitted to Storm cluster is divided into several tasks.

2.3. Apache Flink

Flink [5] is an open source framework for processing data in both real time mode and batch mode. It provides several benefits such as fault-tolerant and large scale computation. The programming model of Flink is similar to MapReduce [4] . By contrast to MapReduce, Flink offers additional high level functions such as join, filter and aggregation. Flink allows iterative processing and real time computation on stream data collected by different tools such as Flume [3] and Kafka [7]. It offers several APIs on a more abstract level allowing the user to launch distributed computation in a transparent and easy way.

2.4. Apache Samza

Apache Samza [9] is an open source distributed processing framework created by Linkedin to solve various kinds of stream processing requirements such as tracking data, service logging of data, and data ingestion pipelines for real time services. Since then, it was adopted and deployed in several projects. Samza is designed to handle large messages and to provide file system persistence for them. It uses Apache Kafka as a distributed broker for messaging, and Hadoop YARN for distributed resource allocation and scheduling. YARN resource manager daemon is adopted by Samza to provide fault tolerance, processor isolation, security, and resource management in the cluster. Samza is based on three layers. The first one is devoted to streaming data and uses Apache Kafka to transit the data flow. The second layer is based on YARN resource manager to handle the distributed execution of Samza processing and to manage CPU and memory usage across a multi-tenant cluster of machines. The processing capabilities are available in the third layer which represents the Samza core and provides API for creating and running stream tasks in the cluster [9]. In this layer, several abstract classes can be implemented by the user to perform specific processing tasks. These abstract classes are implemented with a MapReduce Framework to ensure the distributed processing.

3. Comparison of stream processing frameworks

In this section, the frameworks presented above are compared according to several features (see Table 1) including: data format, types of data sources, programming model, cluster manager, supported programming languages, latency and messaging capacities.

We notice that Spark importance lies in its in-memory features and micro-batch processing capabilities, especially in iterative and incremental processing [2]. Although Spark is known to be the fastest framework due to the concept of RDD, it remains characterized by its low throughput compared to other frameworks, while its micro-batch concept could guarantee the fault tolerance. Flink shares similarities and characteristics with Spark. It offers good processing performance when dealing with complex big data structures such as graphs. Although there exist other solutions for large-scale graph processing, Flink and Spark are enriched with specific APIs and tools for machine learning, predictive analysis and graph stream analysis [1] [14].

	Spark	**Storm**	**Flink**	**Samza**
Data format	DStream	Tuples	DataStream	Message
Data sources	HDFS, DBMS, and Kafka	Spoots	HDFS, DBMS, and Kafka	kafka
Programming model	Transformation and action	Bolts	Actions functions (map,groupby,..)	Mapreduce Job
Programming languages	Java, Scala and Python	Java	Java	java
Cluster manager	Hadoop YARN, Apache Mesos	Zookeeper	Hadoop YARN, Apache Mesos	YARN
Latency	Few seconds	Sub-second	Sub-second	Sub-second
Messaging	Exactly once	At least once	Exactly once	Exactly once
Machine learning compatibility	SparkMLLIB	Compatible with SAMOA API	FlinkML	Compatible with SAMOA API
Elasticity	Yes	Yes	No	No
Sliding windows/Windowing	time based	time based and count based	time based	time based and count based
Auto-parallelization	On demand	Pipelined processing	Pipelined processing	On demand
Streaming query	SparkSQL	No	No	Yes (Samza-SQL API)
Data Partitioning	Yes	No	No	Yes
API	Declaratif	Copositionnel	Declartaif	Copositionnel
Data transport	RPC	RPC	RPC	Kafka

Table 1. Comparison of popular stream processing frameworks

In contrast, resource allocation in Storm is ensured in a dynamic and transparent way. While existing stream processing frameworks implement their own message transport protocol, Samza jobs use a set of named Kafka topics as input/output. Although the low-level one-message-at-a-time model offers some flexibility to Samza, it presents limitations regarding the frequency of produced errors and the automatic optimization. When a broker node fails, the messages located in the file system will be lost and cannot be recovered.

4. Experiments

In this section, we first present our experimental environment and protocol. Then, we discuss the obtained results. More detailed experiments could be found in [8].

4.1. Experimental environment and protocol

All the experiments were performed in a real cluster called GALACTICA[1]. The cluster is composed of 10 machines operating with Linux Ubuntu 16.04. Each machine is equipped with a 4 CPU, 8GB of main memory and 500 GB of local storage. For our tests, we used Flink 1.3.2, Spark 1.6.0, Samza 0.10.3 and Storm 1.1.1. All the studied frameworks have

[1]https://galactica.isima.fr

been deployed with YARN as a cluster manager. For our experimental protocol, we used Twitter4J API[2] to stream tweets that contain the "Big Data" word in real time. Every tweet consists of a JSON file with a set of attributes such as tweet creation date, tweet identifier and user informations. Our experimental protocol consists on executing an Extract, Transform and Load (ETL) routine that (1) extracts tweets using Kafka in order to ensure the same streaming rate while evaluating the studied frameworks, (2) transforms the tweets by keeping only attributes like tweet identifier, tweet content, date, geocoordinate and user informations, and (3) loads the transformed tweets to ElasticSearch. In this work, we studied: (1) the number of messages processed by each framework in a given period, (2) the impact of the size of the message on the number of processed messages, and (3) the resources consumption of the studied frameworks.

4.2. Experimental results

Figure 1 shows that Flink, Samza and Storm have better processing rates compared to Spark. This can be explained by the latency feature. In fact, the latency of Spark about seconds while it is about subseconds in the case of Flink, Samza and Storm (see Table 1).

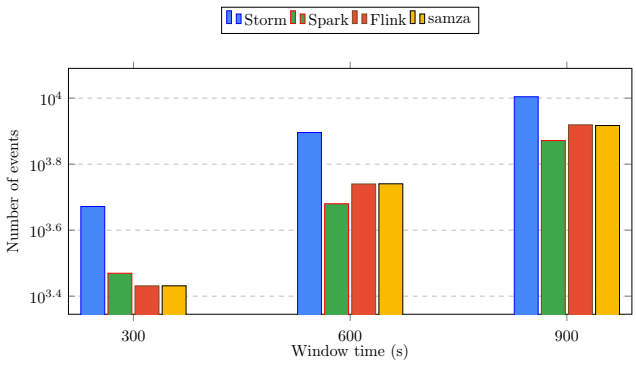

Figure 1. Impact of the window time on the number of processed events (100 KB per message)

In the next experiment, we changed the sizes of the processed messages. We used 5 tweets per message (around 500 KB per message). The results presented in Figure 2 show that Samza and Flink are very efficient compared to Spark, especially for large messages.

CPU consumption

As shown in Figure 3, Flink CPU consumption is low compared to Spark, Samza and Storm. Flink exploits about 10% of the available CPU, whereas Storm CPU usage varies between 15% and 18%. However, Flink may provide better results than Storm when CPU resources are more exploited. In the literature, Flink is designed to process large messages, unlike Storm which is only able to deal with small messages (e.g., messages coming from sensors). Unlike Flink, Samza and Storm, Spark collects events' data every second and performs processing task after that. Hence, more than one message is processed, which explains the high CPU usage of Spark. Because of Flink's pipeline nature, each message is associated to a thread and consumed at each window time. Consequently, this low volume of processed data does not affect the CPU resource usage. Samza exploits

[2]http://twitter4j.org/en/index.html

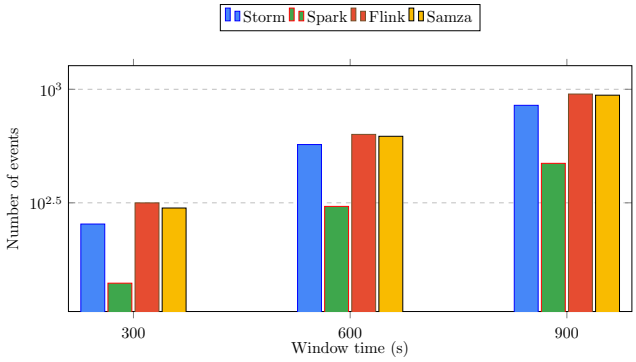

Figure 2. Impact of the window time on the number of processed events (500 KB per message)

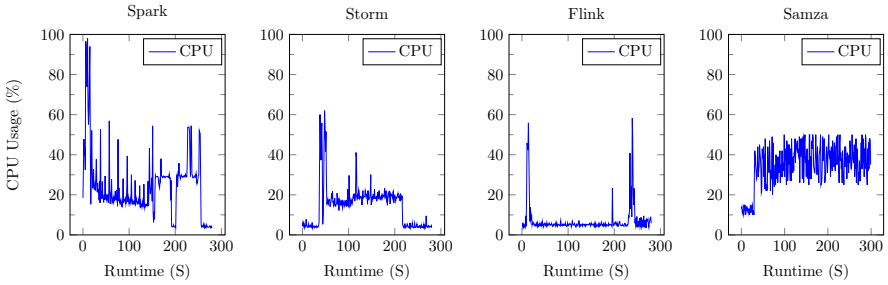

Figure 3. CPU consumption

about 55% of the available CPU because it is based on the concept of virtual cores and each job or partition is assigned to a number of virtual cores. In fact, it deploys several threads (one for each partition), which explains the intensive CPU usage by Samza compared to the other frameworks.

RAM consumption

Figure 4 shows the cost of event stream processing in terms of RAM consumption. Figure 4 shows the cost of event stream processing in terms of RAM consumption. Spark reached 6 GB (75% of the available resources) due to its in-memory behavior and its ability to perform in micro-batch (process a group of messages at a time). Flink, Samza and Storm did not exceed 5 GB (around 61% of the available RAM) as their stream mode behavior consists in processing only single messages. Regarding Spark, the number of processed messages is small.

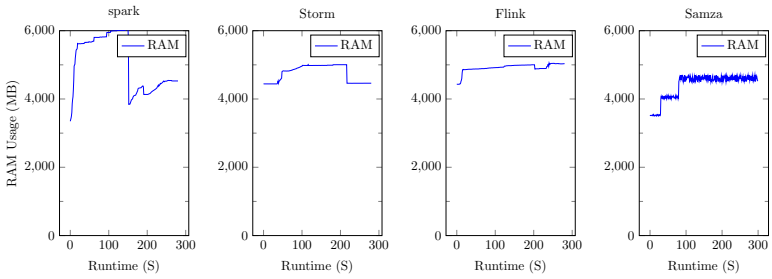

Figure 4. RAM consumption

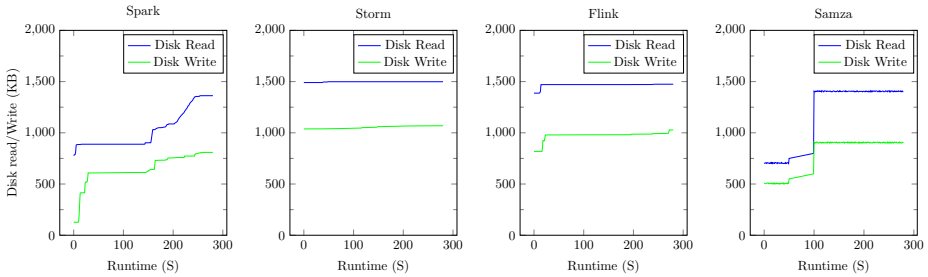

Figure 5. DISC R/W consumption

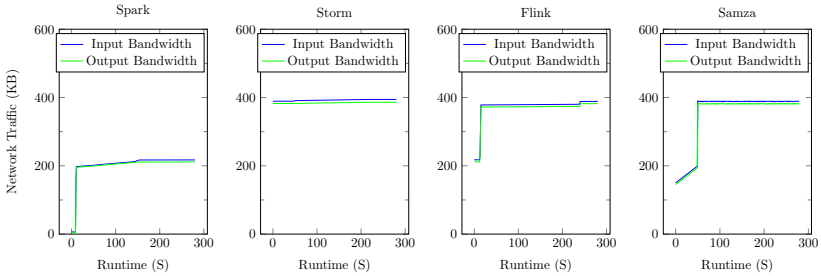

Figure 6. Bandwidth consumption

DISC R/W consumption

Figure 5 depicts the amount of disk usage by the studied frameworks. The curves denote the amount of Read/Write operations. The amounts of Write operations in Flink and Storm are almost close. Flink, Samza and Storm frequently access the disk and are faster than Spark in terms of the number of processed messages. As discussed in the above sections, Spark is an in-memory framework which explains its lower disk usage.

Bandwidth resource usage

As shown in Figure 6, the amount of data exchanged per second varies between 375 KB/s and 385 KB/s in the case of Flink, and varies between 387 KB/s and 390 KB/s in the case of Storm. It is about 400 Mb/s in the case of Samza. This amount is high compared to Spark as its bandwidth usage did not exceed 220 KB/s. This is due to the reduced frequency of serialization and migration operations between the cluster nodes, as Spark processes a group of messages at each operation. Consequently, the amount of exchanged data is reduced, while Storm, Samza and Flink are designed for the stream processing.

5. Conclusion

With the increasing amount of data generated by billions of devices over the world, stream processing becomes a key requirement of big data frameworks. The main goal of the present work is to study and experimentally evaluate the most popular frameworks for large-scale stream data processing. Spark, Storm, Flink and Samza were presented and categorized according to their main features. We also evaluated the performance of the presented frameworks in terms of resource consumption. We mention that this work is a part of our previously published paper [8]. In this work, we focus on the evaluation of streaming frameworks for Big Data. We mainly added a categorization of the studied frameworks based on specific features of stream processing systems. In the future, we will address the velocity of data processing by conducting more experiments on the frequency

and the size of incoming events data.

Acknowledgements

This research was partially supported by the General Direction of Scientific Research in Tunisia (DGRST).

References

[1] Alexander Alexandrov, Rico Bergmann, Stephan Ewen, Johann-Christoph Freytag, Fabian Hueske, Arvid Heise, Odej Kao, Marcus Leich, Ulf Leser, Volker Markl, et al. The stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, 2014.

[2] Fuad Bajaber, Radwa Elshawi, Omar Batarfi, Abdulrahman Altalhi, Ahmed Barnawi, and Sherif Sakr. Big data 2.0 processing systems: Taxonomy and open challenges. *Journal of Grid Computing*, 14(3):379–405, 2016.

[3] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R Henry, Robert Bradshaw, and Nathan Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[4] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[5] Apache Flink. Scalable batch and stream data processing, 2016.

[6] Amir Gandomi and Murtaza Haider. Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management*, 35(2):137–144, 2015.

[7] Nishant Garg. *Apache Kafka*. Packt Publishing Ltd, 2013.

[8] Wissem Inoubli, Sabeur Aridhi, Haithem Mezni, Mondher Maddouri, and Engelbert Mephu Nguifo. An experimental survey on big data frameworks. *Future Generation Computer Systems*, 86:546 – 564, 2018.

[9] Apache Samza. Linkedin's real-time stream processing framework, by riccomini, c, 2014.

[10] Kamran Soomro, Zaheer Khan, and Khawar Hasham. Towards provisioning of real-time smart city services using clouds. In *Proceedings of the 9th International Conference on Utility and Cloud Computing*, pages 191–195. ACM, 2016.

[11] Apache Spark. Apache spark: Lightning-fast cluster computing, 2015.

[12] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M Patel, Sanjeev Kulkarni, Jason Jackson, Krishna Gade, Maosong Fu, Jake Donham, et al. Storm@ twitter. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 147–156. ACM, 2014.

[13] Christos Vlassopoulos, Ioannis Kontopoulos, Michail Apostolou, Alexander Artikis, and Dimitrios Vogiatzis. Dynamic graph management for streaming social media analytics. In *Proceedings of the 10th ACM International Conference on Distributed and Event-based Systems*, pages 382–385. ACM, 2016.

[14] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.